WAIT: Workshop on the Advances in Information Technology, Budapest, 2015

Tonal Art Maps with Image Space Strokes

L. Szécsi and M. Szirányi

Department of Control Engineering and Information Technology, Budapest University of Technology and Economocs, Budapest, Hungary

Abstract

This paper presents a hybrid hatching solution that uses robust and fast texture space hatching to gather stroke fragments, but fits stylized brush strokes over those fragments in image space. Thus we obtain a real-time solution that avoids the challenges associated with hidden stroke removal in image space approaches, but allows for the artistic stylization of strokes exceeding the limitations of texture space methods. This includes strokes running over outlines or behind occluders, uniquely random strokes, and adherence to image space brush properties.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Line and Curve Generation

1. Introduction

Image synthesis methods mimicking artistic expression and illustration styles [3, 13, 14] are usually vaguely classified as *non photo-realistic rendering* (NPR). Hatching is one of the basic artistic techniques that is often emulated in stylistic animation. Hatching strokes should appear hand-drawn, with roughly similar image-space width, dictated by pencil or brush size, but they should also stick to surfaces to provide proper object space shape and motion cues. Both properties must be maintained in an animation, without introducing temporal artifacts [5, 1]. In particular, when surface distance or viewing angle is changing, object-space density of strokes should adapt without the strokes flickering or drifting on the surface, while presenting natural randomness inherent in manual work. When zooming in or zooming out, this means adding and removing strokes gradually. Strokes should be uniform as drawn by the same brush or pencil, but also unique. Overdrawn lines crossing object contour should occur.

In this paper, we introduce a hybrid hatching approach (Figure 1) built on texturing-based hatching, where we fit image-space strokes on hatching lines to provide style options not available just with surface shading.

2. Previous work

Several works proposed the application of *particles* or *seeds* either attached to objects [9, 16], or moving along with the optical flow in image space [17]. Seeds are extruded to textured triangle strips representing hatching strokes in image space. Strokes are obtained by integrating the direction vector field started at seed points or particles [19, 12]. The key problem in these methods is the generation of the world-space seed distribution corresponding to the desired image-space hatching density. This either means seed killing and fissioning [18]—even using mesh subdivision and simplification [2]—, or rejection sampling [16]. These techniques are mostly real-time, but require multiple passes and considerable resources. Compositing these with three-dimensional geometry is challenging: as extruded hatching curves do not strictly adhere to surfaces, depth testing them against triangle mesh objects must be using heavy bias and smooth rejection to avoid flickering.

The visibility problem is solved robustly in *texturing-based approaches* [7], where the hatching pattern is drawn on object surfaces. Uniform screen-space hatching density can be achieved using different level-of-detail textures, either pre-drawn as *Tonal Art Maps (TAM)* [13], or procedurally defined as *Recursive Procedural Tonal Art Maps (RP-TAM)* [15]. As density is locally evaluated at shaded surface points, some strokes are clipped or faded out as they would extend to an area of lower required density. Strokes are also clipped at object silhouettes or parametrization discontinuities, causing deviations in style.

Fitting curves on outlines is a straightforward idea employed both for silhouettes [11] and in sketching [8]. Modeling hatching strokes as curves [6] was also used. We are not aware of any work where en masse curve fitting for hatching strokes was proposed.

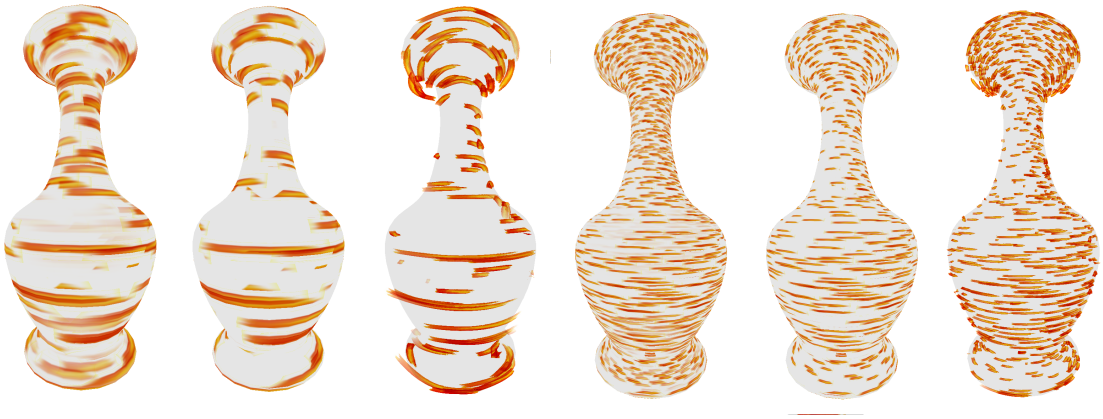

Figure 1: *Texture-space methods must fade (1, 4) or clip (2, 5) strokes for density control. Our solution (3, 6) restores stylistic coherence by fitting new strokes in image space.*

3. Proposed method

In this paper we propose *Tonal Art Maps with Image Space Strokes* (TAMISS), a hybrid technique that combines the robust visibility testing and density control of TAM or RPTAM with the stylistic freedom of image space stroke extrusion. The idea is to assign unique IDs to all TAM strokes, perform rasterization of surfaces with TAM, producing fragments marked with stroke IDs, and fit a curve on each set of fragments sharing the same ID. The curves can be extruded to image space strokes in proper style, while visibility and density control has already been taken care of by TAM.

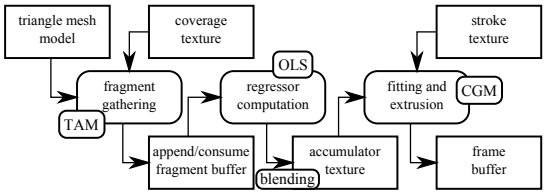

Figure 2: *Pipeline for the proposed method. TAM renders to a list of fragments instead of the frame buffer. Fragments are routed by ID to texture accumulating regressors. For every texel, a curve is fit and a textured, stylized stroke is extruded.*

3.1. Method outline

Figure 2 depicts the algorithm workflow. We need TAM textures storing stroke IDs instead of colors. As multiple strokes may overlap, TAM texels need to contain short lists of stroke IDs. The RPTAM approach already makes use of such textures, called *stroke coverage textures* there. In the first, *fragment gathering* phase, surfaces are rasterized at a moderate resolution. For every surface fragment, as many stroke fragments are generated as there are strokes in the corresponding stroke coverage texel. All fragments store a *global stroke ID*, which contains the ID from the coverage texture, but must be unique over all surfaces. In the next, *regression* step, for every ID, the corresponding fragments are processed, and the best fitting curves are found. Finally, the curves are *extruded* to textured triangle strips and rendered to the frame buffer.

3.2. ID generation

If two distant strokes share the same ID, the curve fitted on their fragments deviates from the original intent and is inconsistent in style. Thus, IDs of strokes visible at the same time should be unique. However, the number of strokes is high with TAM and infinite with RPTAM, and our set of IDs is limited (in our implementation we use just 16 bits). We assume surfaces are decomposed into relatively simple segments, where texture space separation implies image space separation. Segment IDs are part of the global stroke ID. Within segments, it is sufficient for the IDs to be just locally unique, meaning that as we zoom in on the surface, IDs of out-of-view strokes can be reused for new strokes appearing on denser levels. This is straightforward with TAM, but challenging with RPTAM.

For RPTAM's seed sets, seed coordinates (u_i, v_i) are obtained by repeatedly left-shifting recurring binary fractions (u_0, v_0), where the period length is some k^4. Combining the respective bits u_{0j} and v_{0j} into quaternary *crumbs* must give quaternary De Bruijn sequences (referred to as "uniform cycles" in [15]). When shading surfaces with RPTAM, the seed sets are tiled indefinitely in texture space, at a scale appropriate for the level of detail. At a surface point, first the appropriate texture scale is found, then the position within the tile of seeds is computed, and relevant seeds (read from the coverage texture) are processed. As a by-product of the computation, the 2D index of the seed tile is available. The concatenation of this 2D tile index and (u_i, v_i) specifies the loca-

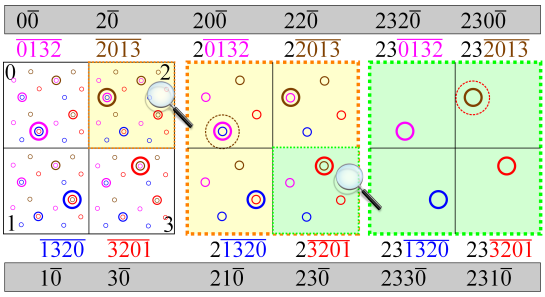

Figure 3: *Infinite positional IDs for RPTAM seeds with $k = 1$ and De Bruijn sequence* 0132*, written as quaternary numbers, constructed by concatenating the tile index and the recurring pattern. Vincula denote indefinite repetition. In the gray boxes the* twisted *versions of the IDs are shown.*

tion of the seed instance within texture space as two binary fractions, providing a unique, but infinitely long *positional ID* for the stroke. Top-level positional stroke IDs are the recurring binary fractions themselves. On more detailed levels, the periods of the binary fraction start after more tile index bits. Figure 3 depicts the ID scheme for the first three detail levels, zooming in to subtile 2 and then to 23. Note that $\overline{2013} = 20\overline{132}$ and $\overline{23201} = 232\overline{013}$.

In RPTAM, a stroke that appears early is guaranteed to stay on later levels, and in this scheme it receives the same position-dependent ID on any level, no matter how many bits come from the tile index. It is crucial that we keep this property as we try to reuse IDs for high-detail strokes distant from each other. Otherwise, parts of strokes crossing detail level boundaries would have different IDs, and drawn as two strokes instead of one.

Let us now consider the IDs as quaternary numbers. Because the crumbs in the recurring part form De Bruijn sequences, where every pattern of length k appears exactly once, any k crumbs can be used to predict the next one. Now let us recode all IDs so that we always write the difference $\pmod 4$ between the actual crumb and the prediction. We call the result the *twisted ID*, which is still a unique, infinitely long ID, but recurring crumbs that do not hold extra information are now zeroes. In Figure 3, 0 should follow 2, so 0 is subtracted from the second crumb. In the next level, 2 should follow 3, so 2 is subtracted from the third crumb. In the recurring part, predictions are always accurate, and the twisted crumbs are zeroes.

Let m be the length (in crumbs) of IDs we can store. Let us consider a single stroke with a twisted ID of length m, plus trailing zeroes. As we zoom in to the next level, we get four strokes, one of them coinciding with the original, with a zero crumb at position $m+1$. Those IDs for which the first crumb was different are far away in the texture, and are therefore presumably zoomed out of view. Thus, to store the $m+1$ crumb, we need to reuse the first one. If we add the new crumb to the first one $\pmod 4$, then the four new strokes get different IDs, but the coinciding one keeps its ID from the previous level. This mechanism can be generalized to the entire sequence: we just add m length chunks of the twisted ID crumbwise $\pmod 4$, to get locally unique IDs of length m.

3.3. Regression

We rasterize the surfaces with appropriate TAM or RPTAM shaders to gather stroke fragments. Fragments are stored with their x_i, y_i screen space coordinates, and t_i values. Parameter t_i specifies where the fragment appears on the stroke. In RPTAM, it is available as the *stroke space coordinate*, for TAM, it must be stored in the coverage texture. We denote the vector of powers of t_i as

$$\mathbf{t}_i = \left(1, t_i, t_i^2, t_i^3\right)^{\mathrm{T}}.$$

Given n fragments of a stroke, we need to find coefficients of the curve equation. As we use cubic curves, the parametric curve equation has the form

$$\mathbf{r}(t) = \left(\mathbf{c}_{\mathrm{x}}^{\mathrm{T}} \cdot \mathbf{t}, \mathbf{c}_{\mathrm{y}}^{\mathrm{T}} \cdot \mathbf{t}\right), \quad \text{with} \quad \mathbf{t} = \left(1, t, t^2, t^3\right)^{\mathrm{T}},$$

where $\mathbf{c}_{\mathrm{x}}$ and $\mathbf{c}_{\mathrm{y}}$ are column vectors of coefficients.

Finding $\mathbf{c}_{\mathrm{x}}$ and $\mathbf{c}_{\mathrm{y}}$ are *linear regression* problems, that we can solve using the *Ordinary Least Squares* method. Using the explicit formula by Hayashi [4], we obtain the linear system

$$\mathbf{b} = \mathbf{A} \cdot \mathbf{c}, \quad \text{with} \quad \mathbf{A} = \sum_{i=0}^{n-1} \mathbf{t}_i \cdot \mathbf{t}_i^{\mathrm{T}}, \tag{1}$$

and either $\mathbf{b} = \sum_{i=0}^{n-1} \mathbf{t}_i \cdot x_i$ and $\mathbf{c} = \mathbf{c}_{\mathrm{x}}$, or $\mathbf{b} = \sum_{i=0}^{n-1} \mathbf{t}_i \cdot y_i$ and $\mathbf{c} = \mathbf{c}_{\mathrm{y}}$.

Solving the system for $\mathbf{c}$ by directly inverting $\mathbf{A}$ is feasible, but it is not the most efficient or stable option, as $\mathbf{A}$ may be singular or close to singular. Matrix $\mathbf{A}$ is *positive definite*, as for any vector $\mathbf{p}$,

$$\mathbf{p}^{\mathrm{T}}\mathbf{A}\mathbf{p} = \sum \mathbf{p}^{\mathrm{T}} \cdot \mathbf{t}_i \cdot \mathbf{t}_i^{\mathrm{T}} \cdot \mathbf{p} = \sum \left|\mathbf{p}^{\mathrm{T}} \cdot \mathbf{t}_i\right|^2 > 0.$$

Therefore, the iterative *conjugate gradient method* [10] (CGM) can be applied, which delivers the pseudo-inverse solution even for singular matrices. In our experience, using 32-bit floating point numbers, performing the theoretically required four iterations is sufficient. Using solutions from previous frames as iteration starting points is therefore not worth the storage lookup time. Note that, in theory, any other method of solving the linear system is applicable, including *singular value decomposition* (SVD). However, CGM translates

to four multiplications of 4×4 matrices and a few four-element vector operations, making it both efficient and easy to implement on the GPU.

The stroke may be only partially visible. We find the useful parameter range of the curve as

$$[t_{\min}, t_{\max}] = \left[\min_i t_i, \max_i t_i \right].$$

However, parts of strokes may be hidden, resulting in discontinuous fragment blocks. This gives us a third variable to perform regression on: visibility. In equation 1, $\mathbf{b} = \sum_{i=0}^{n-1} \mathbf{t} \cdot v_i$ and $\mathbf{c} = \mathbf{c}_\mathrm{v}$, where visibility v_i is one at every fragment, and we have no data points for regression where v_i would be zero. Such data points, however, can easily be added by assuming they are evenly distributed in $[t_{\min}, t_{\max}]$. Given that v_i is zero at these points, $\mathbf{b}$ does not change, but $\mathbf{A}$ must include their contribution. For n new points, this can be computed analytically as

$$\mathbf{D} = n \int_{t_{\min}}^{t_{\max}} \mathbf{t} \cdot \mathbf{t}^\mathrm{T} \mathrm{d}t, \quad \text{yielding} \quad d_{i,j} = n \frac{t_{\max}^{i+j-1} - t_{\min}^{i+j-1}}{i+j-1}.$$

Solving the system $\mathbf{b} = (\mathbf{A} + \mathbf{D}) \cdot \mathbf{c}_\mathrm{v}$, we obtain visibility function $v(t) = \mathbf{c}_\mathrm{v}^\mathrm{T} \cdot \mathbf{t}$, which can be thresholded to obtain visible stroke segments. A cubic fit on the visibility works as long as there are no more than two visible segments. More complex cases are rare, and easily pass as artistic inaccuracies. Extending the regression problem to a higher order fit is trivial, but a shader implementation would be much less elegant.

4. Implementation

The solid geometry depth is laid down first, so that only visible surfaces are rendered. The TAM or RPTAM implementation needs to be modified slightly to stream fragments into a buffer. This can be accomplished in a fragment shader without render target output, but writing to an appendable random access GPU buffer. The list of fragments can be rendered as a vertex buffer of point primitives, with textures as render targets, where every texel corresponds to a possible ID. The vertex shader positions the point primitives by ID, and after rasterization blending is used to add $\mathbf{t}_i \cdot \mathbf{t}_i^\mathrm{T}$, $\mathbf{t} \cdot x_i$, and $\mathbf{t} \cdot y_i$ to the texels. Values $t_{\min}$ and $t_{\max}$ are found with maximum blending.

Alternatively, without using the intermediate fragment buffer, the values can be aggregated using atomic operations. We found this adequate for lower order fitting, but as atomics only work with fixed-point representation, the floating point range provided by blending is indispensible for summing higher powers of t_i.

In the final pass, for every possible ID, a dataless point primitive is rendered. The geometry shader fetches the coefficients from the aggregate textures, and solves the regression equations using CGM. Strokes with just a few fragments are filtered out. The shader also segments the curve by visibility, and extrudes it to a triangle strip in screen space. Any additional stylization like per-stroke randomization can be performed here.

obj	△	∼	ID	OLS	CGM	E&R
vase	27k	1k	2.36	3.06	0.04	0.11
vase	27k	4k	2.78	2.96	0.19	0.32
knight	90k	4k	2.57	2.86	0.31	2.01
head	193k	4k	2.81	2.78	0.23	0.51
head	193k	16k	2.99	2.88	0.91	2.04

Table 1: *Performance on different objects (with the number of triangles and hatching strokes specified) at* 1920×1200*. Rendering times are given in ms for the modified texture-based hatching shader that computes the IDs (ID), aggregation of fragments for regression (OLS), cubic curve fitting (CGM), and final stroke extrusion and rendering (E&R).*

4.1. Results and conclusions

We measured performance on an NVIDIA GeForce GTX 780, with 1920×1200 full-screen resolution (Table 4.1, Figure 4). Compared to single pass texturing with RPTAM, TAMISS takes about five times as long for simple geometries, but performs still well over 100 FPS in full screen. The three most expensive operations are computation of the twist IDs, summation of fragments, and rasterizing the final strokes. The solution of the regression equations with CGM is negligible. For more complex scenes, the overhead remains constant. Performance depends heavily on the number of strokes, but the meaningful range is quite limited, with 32k being entirely sufficient.

Figure 4: *The* knight *and* head *test objects.*

As fitting is performed independently in every frame, any rasterization or numerical inaccuracies may manifest as jittering in stroke positions. This can be alleviated by using a

higher resolution for fragment gathering, resulting in more fragments to process. This cost could be amortized by averaging aggregate fragment data over several frames instead.

While higher order fitting is certainly possible, even a quartic solution would need approximately doubled computation and storage requirements. We think this would be for no practical gain, as the underlying shape is adequately represented by cubics. This does not mean that strokes could not have additional stylization like waves or zigzags, even on a stroke-by-stroke basis. Thus, our method does not limit visual style compared to TAM, but allows for more stylistic freedom by decoupling stroke positioning and stroke style.

The main limitation of our method is that it needs an overlap-free UV mapping, oriented on object features, and separated into spatial-coherence-preserving charts. Such an UV mapping is never readily available, and we have yet to propose an automated solution, or show that an existing one like *lapped textures* as in [13] can be adapted.

Acknowledgements

This work has been supported by OTKA PD-104710 and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

1. Zainab AlMeraj, Brian Wyvill, Tobias Isenberg, Amy A Gooch, and Richard Guy. Automatically mimicking unique hand-drawn pencil lines. *Computers & Graphics*, 33(4):496–508, 2009.
2. Derek Cornish, Andrea Rowan, and David Luebke. View-dependent particles for interactive non-photorealistic rendering. In *Graphics interface*, volume 1, pages 151–158, 2001.
3. Paul Haeberli. Paint by numbers: Abstract image representations. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 207–214. ACM, 1990.
4. F Hayashi. *Econometrics*. Princeton University Press, 2000.
5. Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. Hatching by example: a statistical approach. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 29–36. ACM, 2002.
6. R.D. Kalnins, L. Markosian, B.J. Meier, M.A. Kowalski, J.C. Lee, P.L. Davidson, M. Webb, J.F. Hughes, and A. Finkelstein. Wysiwyg npr: Drawing strokes directly on 3d models. *ACM Transactions on Graphics*, 21(3):755–762, 2002.
7. Hyunjun Lee, Sungtae Kwon, and Seungyong Lee. Real-time pencil rendering. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 37–45. ACM, 2006.
8. John P Lewis, Nickson Fong, Xie XueXiang, Seah Hock Soon, and Tian Feng. More optimal strokes for npr sketching. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 47–50. ACM, 2005.
9. Barbara J Meier. Painterly rendering for animation. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484. ACM, 1996.
10. Jorge Nocedal and Stephen J Wright. *Conjugate gradient methods*. Springer, 2006.
11. JD Northrup and Lee Markosian. Artistic silhouettes: A hybrid approach. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 31–37. ACM, 2000.
12. Afonso Paiva, Emilio Vital Brazil, Fabiano Petronetto, and Mario Costa Sousa. Fluid-based hatching for tone mapping in line illustrations. *The Visual Computer*, 25(5-7):519–527, 2009.
13. Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 581–581. ACM, 2001.
14. Thomas Strothotte and Stefan Schlechtweg. *Non-photorealistic computer graphics: modeling, rendering, and animation*. Elsevier, 2002.
15. L Szécsi and M Szirányi. Recursive porcedural tonal art maps. In *WSCG 2014 Full Papers Proceedings*, pages 57–66. Union Agency, 2014.
16. Tamás Umenhoffer, László Szécsi, and László Szirmay-Kalos. Hatching for motion picture production. In *Computer Graphics Forum*, volume 30, pages 533–542. Wiley Online Library, 2011.
17. David Vanderhaeghe, Pascal Barla, Joëlle Thollot, and François Sillion. Dynamic point distribution for stroke-based rendering. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, pages 139–146, 2007.
18. Andrew P Witkin and Paul S Heckbert. Using particles to sample and control implicit surfaces. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 269–277. ACM, 1994.
19. Johannes Zander, Tobias Isenberg, Stefan Schlechtweg, and Thomas Strothotte. High quality hatching. In *Computer Graphics Forum*, volume 23, pages 421–430. Wiley Online Library, 2004.